"A Tale of Two Provers":

A Comparison of Dependent Haskell and F*

Xinyue Zhang
Advised by Richard A. Eisenberg
March, 2018

Submitted in Partial fulfillment of the requirement for Bachelor of Arts in Computer Science at Bryn Mawr College

Abstract

This is the abstract.

To my dearest grandpa Zhaomin,

who has supported me in his own way toward this bachelor's degree.

Acknowledgements

My sincere gratitude goes first to my advisor, Professor Richard Eisenberg, whose unwavering passion and patient guidance have kept me constantly engaged in my research. I am very grateful to Professor Dianna Xu and Professor Deepak Kumar, whose insight, advice and encouragement have been invaluable to me over these four years. I must thank every one from the Computer Science departments at Bryn Mawr and Haverford Colleges, and my special thanks goes to Professor Kristopher Micinski.

I truly appreciate all the companionship from my fellow PL enthusiasts. Thanks to the F* developers, Nikhil Swamy and Tahina Ramananandro for their timely advice. Thanks to all members of the PL Club at UPenn, especially Yishuai Li, for their inspiration along the way. Special shout out to Rachel Xu, Kevin Liao, and Divesh Otwani, for together forming such an amazing PL squad at Bi-Co.

I wouldn't be able to achieve thus far without the support of my family and friends. To my parents, thanks for believing in me every single moment and encouraging me in all of my pursuits. To my dearest grandpa Zhaomin, no words can express how much I love you and miss you. I couldn't imagine how happy and proud you would be if you were to attend my graduation. To my friends, thank you for all the calls, texts, postcards, hugs, visits, advice, and for just being there. I own a debt of gratitude to Jordan Henck and Calla Carter, your friendship is more than I could ever ask. Finally I would like to thank my peer reviewers: Kara Breeden, Nora Broderick, Stephanie Cao, and Trista Cao. I am very grateful for their valuable feedback about my drafts.

Contents

1 Introduction

> *Testing can show the presence of errors, but not their absence.*
>
> – Edsger Wybe Dijkstra

1.1 Motivation

Programming languages are powerful tools used to design and build complex computer software in order to perform various tasks. Yet complex software programs often have bugs that prevent them from functioning as expected. Almost everything is dependent on software, from smart devices to vehicles, and to national defense systems. Most of the time bugs cause small problems such as an application crash. However, in many cases, tiny errors in a program cause catastrophic if not life threatening consequences such as a financial crisis or a plane crash. Therefore, various research must be done to increase our confidence on the correctness of a program.

Through systematic testing, it is possible to find some incorrect program behaviors, assuming that it is easy or even possible to check the correctness of a program output. Yet exhaustive testing like this is often limited to a given set of test cases, making it unsuitable as a general solution. As Edsger Dijkstra stated, "testing can show the presence of errors, but not their absence", testing cannot guarantee the correctness of a program [3]. Other methods for error checking exist but we are specifically interested in formal verification.

Formal verification uses formal methods, such as logic, to ensure the correctness of software with respect to a defined or inferred specification. We say software is correct if it meets its specification, which defines all its intended behaviors. There are various techniques towards formal verification, each with its own advantages and disadvantages. For example, model checking is one common lightweight formal method [4]. Yet, type system design is by far the most widely-adopted and well-formed method [14]. A type system is a mechanism that enforces rules, known as the type rules, on programs to help reason about them and to prevent specified errors. A **type** classifies values and expressions into categories that share certain properties. For example, we say that `true` and `false` are both of type `Boolean`. Since the type rule on booleans does not support the addition operation(`+`), we can catch invalid operations such as `true + false`. In general, a well-defined type system may be applied to programming languages to help verify, at compile time, that all programs written perform as intended. Therefore, the desire for a stronger type system to allow more programs to be typed is a major focus in the programming languages research.

Programming languages like Coq [5], Agda [13], and Dafny [11] have been studied for a long time and are relatively well understood. They together laid down a solid foundation for program verification. On the other hand, researchers on functional programming languages, like Haskell and ML, gradually start to support verifications with dependent types, resulting in the design of verification-oriented Dependent Haskell and F* [7, 15].

Comparing to Coq, Agda and Dafny, Dependent Haskell and F* are easier to read, program and maintain.

Dependent Haskell and F* each presents a unique and effective program verification technique. They are similar in that they both aim towards a verification-oriented and general-purpose programming language. They are, however, noticeably different in their approaches in verifying potential divergent functions. Dependent Haskell does not require termination checking, whereas F* categorizes divergent functions into a special `Dv` effect (See section 2.5.4 for details). Most dependently typed verifiers require compile-time termination checking to ensure the correctness of proofs [7]. However, as Alan Turing proved via the Halting Problem, given an arbitrary program and its inputs, checking whether this program will terminate or loop forever is an undecidable problem. Yet functions that diverge or whose termination are hard to prove are very common in practice. Therefore, we are interested in exploring how to better express and reason about them.

This thesis aims to introduce two verification-oriented languages: Dependent Haskell and F*, present their unique type system designs, and provide a detailed comparison between the languages.

1.2 Contributions

We offer the following contributions:

- To our knowledge, this is the first work to compare Dependent Haskell and F*, two verification-oriented yet general-purpose functional programming languages. Specifically, we presented the following original results:

  1. We collected examples to compare the user interface of Dependent Haskell and F*, focusing on syntactic support, library and documentation support, and proof complexity (Chapter 4.1).
  2. We explored and analyzed the strengths and limitations of Dependent Haskell's and F*'s approaches towards potentially non-terminating functions (Chapter 4.2).
  3. We highlighted the preferable use cases for programmers to choose between Dependent Haskell and F*, and we provided suggestions to both communities for potential future research directions (Chapter 5).

  We expect our research to set up an initial step for potential collaborations between the two communities in the future, and inspire thoughts on making trade-offs between program reliability and verification support when reasoning about functions whose termination is hard to prove.

- We classified and illustrated the two verification techniques used in Dependent Haskell and F*. Although the contents presented in Chapter 2 are not original, the thorough and friendly review is a true contribution of this thesis. We expect anyone with 3 - 4 years of programming experience to be able to learn some functional

programming concepts and to experience some program verification principles.

We believe that our work will elicit more interest in programming languages research among undergraduate Computer Science majors, given the strong excitement received from the underclassmen.

2 Background

2.1 From Imperative to functional programming

Most programmers come from a Java or C/C++ background [1]. These languages are designed to primarily support imperative or procedural programming. In the imperative style of programming, a developer writes code that details each step for computers to execute. These imperative programs often have side effects that modify program inputs or global variables. Any function modifying global states in execution is known as stateful. For example, global variables can be modified in any function in scope. Functional programming, as its name suggests, models a problem as a set of functions to be executed, detailing the inputs and outputs for each function. Some common functional programming languages are Haskell, ML, Lisp and Scheme. Similar to the mathematical functions introduced in high school algebra, functions in these languages are abstracted to operate on any structured data types, such as boolean, string, list, etc. Functional programs are usually executed by evaluating expressions and typically avoid mutating program states. Functional programming languages can be typed, as in Haskell and ML, or untyped as in Lisp and Scheme. A language is said to be typed if it has a static type system where all values must be classified into types and checked with the system at compile time.

One major difference between imperative and functional style of programming is how they handle program iterations. Imperative programming languages usually support both loops and recursion, however functional programming languages depend highly on recursion and higher-order functions, like `map`, `filter` etc. For example, if one wants to double every element of the list `[1,5,2,4,3]`, one can implement in Java, an imperative programming language as shown below:

```
public void double(int[] arr) {
    for(int i = 0; i < arr.length; i++) {
      arr[i] = 2 * arr[i];
    }
}
```

Notice that in this `double` example, the input list `[1,5,2,4,3]` is directly modified after the execution of the function to the result `[2,10,4,8,6]`.

On the other hand, the same operation can be defined in Haskell in functional style as seen in the code below,

```
double :: [Int] -> [Int]
double []       = []
double (x : xs) = (2 * x) : double xs
```

In the `double` example above, the input is specified as `[Int]`, denoting a list of integers, and output also as `[Int]`. Specifically, when applying `double` to the input list as in `double [1,5,2,4,3]`, we multiply the first element in the input list by 2 and prepend the result to the recursive call on the remaining list until we reach the empty list.

2.2 Program Specification with Dependent and Refinement Types

The starting point of any program verification is to specify program properties. There are two main ways to specify a program, through dependent type and refinement type. A dependent type, whose definition is predicated upon a dynamic value, is a type that captures additional information of the traditional types. As such, dependently-typed programming languages enable programmers to express more detailed specifications with types and to perform more powerful verifications through type checking. On the other hand, a refinement type is a type that satisfies a given logical predicates. It helps to make specifications more intuitive.

To illustrate these definitions in more detail, consider the example of defining a list. In imperative typed programming languages like Java or C/C++, elements in a list must all share a homogeneous type, such as the boolean type. Take Java as an example, an array of length 5 will be constructed as `new boolean[5]`. To specify a list v carrying its own length in verification-oriented programming languages, one can predicate the type of this list of booleans v on its length 5, as in the type `Vec 5 Bool`. The list v's type `Vec 5 Bool` is said to be dependently-typed as it is indexed on the number 5 of another type `Nat` (the type `Nat` represents the natural numbers, defined to be non-negative integers). Similarly, this list of booleans of length 5 can be specified through logical predicates using refinement types. Recall that refinement types specify a logical formula that the types have to additionally satisfy. So one must first define a function `len` calculating the length of a list and then apply it in the predicate to produce the refinement type `v:List Bool{len v = 5}`. The logical formula `len v = 5` is called the predicate of the refinement type, and `list` is a built-in data type in F*. Both dependent types and refinement types are commonly applied in program verifications. Dependent Haskell only supports dependent types, but F* supports both dependent types and refinement types. We will compare them in detail in our result section.

2.3 Dependent Haskell

2.3.1 Introduction to Haskell

Haskell is a strongly typed purely functional programming language. As a pure functional programming language, each function in Haskell only takes in some inputs and returns an output after some operations, without modifying any unspecified program states. Unlike many functions (which should really be called procedures) in an impure language, functions in Haskell do not mutate variables or handle errors. They also do not read or write to standard inputs or outputs, nor do they connect to sockets to communicate with the outside world. This seemingly limiting feature ensures that each execution of the function always returns exactly the same result, which allows formal proofs on functions and also composing functions to perform more complex operations.

Haskell is a typed language, where every variable's type is determined at compile time and is checked before performing any operations. In addition, Haskell's type system enforces strict typing rules on functions, making Haskell a strongly-typed language.

2.3.2 Dependently-Typed Haskell

For a long time, Haskell researchers have strived to add dependent types to Haskell, as dependent types introduce much more precise expressions of program specifications. Although Haskell doesnt yet support dependent types in full, several extensions to its current type system have made exceptional progress.

In 2012, Eisenberg and Weirich introduced the `singletons` library that simulates dependently-typed programming in Haskell. Through dependent types, programmers can reason about programs through computations on the indexed types. For example, when concatenating two lists of booleans, each of length `m` and `n`, the output list can be specified to have type `Vec (Plus m n) Bool`, indexed on the type variable `Plus m n` representing the sum of the two lengths of the original lists. However, Haskell enforces two separate scopes in programming functions – the terms executed at run time and the types checked at compile time. Functions in these two scopes cannot be used interchangeably, so singleton types are necessary to bridge the gap.

Beyond Singletons, Gundry and Eisenberg both conducted research on dependently-typed Haskell using GADTs and type families (Detailed definitions and examples are provided in section 2.4 Program Verification) [9, 7]. Eisenberg, extending Gundrys work, enables Haskell to support full-spectrum dependently typed programming and is partially implemented in the Glasgow Haskell Compiler (GHC) 8.0 [7]. Once implemented in full, Eisenberg's work would provide a backward compatible type system design to support true dependent types in Haskell [7]. In this thesis, Dependent Haskell refers to the current implementation in GHC 8.0.

2.4 Program Verification in Dependent Haskell

After a brief overview of Haskell and Dependent Haskell, this section focuses on Dependent Haskell's verification techniques. The section is structured in a tutorial manner, where concepts are introduced sequentially and gradually build up to two program verification examples as `insert` and `concat`.

2.4.1 Algebraic Data Types

Type-system-based program verification is mainly focused on types. Beyond the primitive types, `Int`, `Bool` etc, which are defined in the standard library, a programmer can also define one's own types. Similar to the C structs, new types can be introduced through the `data` keyword.

For example, a list of elements of type `a` is defined recursively in Haskell as:

```
data List a = Nil | Cons a (List a)
```

The `List` type has two value constructors `Nil` and `Cons`. Each constructor specifies a value that the type could have. Specifically, `Nil` represents the empty list and `Cons` combines one more element to an existing list to form a new list. For example, suppose we have a list of integers `lst1 = [2,3,4,5]`, the expression `Cons 1 lst1` represents the new list `lst2 = [1,2,3,4,5]`. The element `1` that is add into the list `lst1` is called the head of `lst2`, and the rest of the list is called the tail of `lst2`. The symbol | represents logical or, so the `List` type defined above can have a value either an empty list or a list of elements of type `a`. The type `a` could represent any type such as `Int`, `Bool` etc, as long as all the `a`s are matched consistently. In general, any type defined with value constructors is called an algebraic data type (ADT).

Similarly, one can formally define the natural numbers recursively in Dependent Haskell as:

```
data Nat = Zero | Succ Nat
```

where `Zero` represents the smallest natural number 0, and `Succ` represents all the remaining nonzero natural numbers. For example, 1 is expressed as `Succ Zero` and 2 is expressed as `Succ (Succ Zero)` etc.

It is also worth mentioning that the above ADT definition in fact declares two entities: a type level `Nat` inhabited by the terms `Zero` and `Succ`, and a kind level `Nat` inhabited by the types `'Zero` and `'Succ` [7]. Kinds can be understood as the type of types, which classify types, as how types classify terms. In 2012, Yorgey et al. introduced the datatype promotion mechanism that automatically promotes datatypes up a level to kinds and the data constructors to type-level data [16]. Hence, the kind definition `Nat` can be

automatically derived from the original datatype declared above.

2.4.2 Generalized Algebraic Data Types

It follows that, by using dependent types, one can encode the length of a list into its type as a natural number, by parameterizing the list by its length and its element type. Below we introduce the dependently typed vector definition in Dependent Haskell:

```
data Vec :: Nat -> Type -> Type where
  Nil  :: Vec Zero a
  Cons :: a -> Vec n a -> Vec (Succ n) a
```

Instead of a simple listing of value constructors, the definition of `Vec` extends that of `List` by explicitly clarifying the type signature for each constructor. The `Vec` datatype is defined to take in a natural number to specify the length of a vector and the type of each element in the vector, resulting in a new type of kind `Type`. Notice that the `Vec` type has the same two value constructors `Nil` and `Cons` as in the `List` type definition above. The constructor `Nil` has type `Vec Zero a`, meaning that it is an empty list of length `0` and each element has type `a`. Again, this type `a` can represent any type in the Haskell type system. The second constructor `Cons` is recursively defined. It adds a new element of type `a` to an existing list of length `n` and result in a new list of length `n+1`. The above definition on `Vec` is called a Generalized Algebraic Data Type, or a GADT. Specifically, pattern matching on `Nil` gives us type `Vec Zero a` because there is no element in an empty list. Similarly, pattern matching on `Cons x xs` gives us type `Vec (Succ n) a`, denoting that the new list has one more element than before. In general, a GADT allows programmers to get specific information about types thorough patterning matching on each value constructor.

2.4.3 A Simple Verification Example

With the introduced concepts as parameterized dependently-typed GADT `Vec`, we begin with a simple example on how program verification works in Haskell.

Below is a definition of the `insert` function in Dependent Haskell to insert an element to a sorted list. To make our code more readable, we introduce the symbol `:>` to replace the `Cons` operator introduced before. From this section on, we will denote the list `Cons x xs` as `x :> xs`.

```
insert :: Ord a => a -> Vec n a -> Vec (Succ n) a
insert elm Nil       = elm :> Nil
insert elm (x :> xs) = if elm <= x then elm :> x :> xs else x :> (insert elm xs)
```

Elements in the list of type `a` are all comparable, since we define `a` to belong to the `Ord` type class for all types that have an ordering. Also, the above implementation uses GADT pattern matching to classify which case to apply. Specifically, `Nil` is to pattern

match the original input list with the empty list, and `(x :> xs)` is to pattern match the original input list with at least an element. The above insert algorithm may be understood imperatively as follows: the element `elm` is inserted to the list if the list input is originally empty. Otherwise, as the list contains ordered elements, one may compare the value of `elm` with the smallest element `x` in the original nonempty list, and either recursively insert the new element into the original list or directly prepend the element to the front of the list. Notice that when the above `insert` algorithm is repeatedly applied, the eventual output list will be sorted.

Similar to writing test cases, program verifications also focus on the specification of each program. For the `insert` function, the length of the list is incremented by `1` every time one inserts a new element. Therefore, the return type for function `insert` can be defined as `Vec (Succ n) a`, encoding the length incrementation in the returned GADT.

After encoding the verification condition in the type level, we will explain step by step how the actual verification is carried out. Firstly, we verify that adding an element to an empty list results in an list of length `1`.

```
insert elm Nil = elm :> Nil
```

Recall that through pattern matching, the `Nil` list will have type `Vec Zero a`. Then, the proof is obvious since we can directly apply our definition of the cons operator `:>` with type `a -> Vec Zero a -> Vec (Succ Zero) a`. It follows that the result list `ele :> Nil` has type `Vec (Succ Zero) a`, representing a list of length `1` with elements of type `a`.

Secondly, we verify that adding an element to a list of length `n > 0` results in a list of length `n + 1`.

```
insert elm (x :> xs) = if elm <= x then elm :> x :> xs else x :> (insert elm xs)
```

Specifically, following the definition in the last line of the above `insert` function, we check if the element `elm` is less than the first(i.e. smallest) element `x` in the sorted input list and separately consider the two cases as follows. Notice that in both cases, the list `x :> xs` has type `Vec n a` from pattern matching.

Case 1: *`elm <= x`*

Let us first consider the case when the inserted element `elm` is less than or equal to the smallest element `x` in the sorted input list. From the definition, we obtain the result `elm :> x :> xs` where `x :> xs` has type `Vec n a`. We can directly follow the definition of the cons operator `:>` and conclude that the result list `elm :> (x :> xs)` has type `Vec (Succ n) a`, with one more element than the input list.

Case 2: *`elm > x`*

Finally, we consider the last case when the inserted element `elm` is greater than the

smallest element `x` in the sorted input list. We know from pattern matching that the input list `x :> xs` has `n` elements. Consider the tail of the list, i.e. `xs`. If we let `xs` have length `m`, then we know `n = m + 1`. Thus, we want to check that if the input list has type `Vec (Succ m) a`, then the resulting list `x :> (insert elm xs)` has type `Vec (Succ (Succ m)) a`. Since we know by inductive hypothesis that `xs` has type `Vec m a`, we can conclude that `insert elm xs` has type `Vec (Succ m) a`. As a result from the definition of the cons operator, the result list has type `Vec (Succ (Succ m)) a`, which is what we want to verify.

2.4.4 Type Family

To perform some more involved computations at type level, such as addition or multiplication, Dependent Haskell introduces what is known as a type family. For example when concatenating two `Vec`s, one needs to verify the specification such that the resulting `Vec` has length equal to the sum of the two concatenated `Vec`s.

Type family is a function that returns a type. It gives us an extended ability to compute over types and is the core of type-level computations. For example, to add two elements of type `Nat` together, we define the type family in Dependent Haskell as:

```
type family Plus (a :: Nat) (b :: Nat) :: Nat where
  Plus Zero b      = b
  Plus (Succ a') b = Succ (Plus a' b)
```

The implementation of `Plus` is exactly the same as the ordinary `(+)` implementation in the term level, except that `Plus` is in the type level. As a type level function, the type family `Plus` can be used as a predicate to dependent types.

The actual implementation is separated into two cases by applying pattern matching on the first summand of `Plus`. If the first summand `a` is `Zero`, then we define our base case such that any sum of `Zero` and another natural number `b` is just the other natural number `b`. To put it in algebraic form, we have `0 + b = b` for all natural number `b`. Otherwise, if the first summand is a non-zero natural number, represented as `Succ a` where `a'` is the natural number one less than `a`, then one recursively applies `Plus` on the smaller natural number `a'` and the other natural number `b` to obtain the sum and add one to obtain the result. Equivalently, in algebraic equations, if `a = 1 + a'` then the operation is essentially `a + b = (1 + a') + b = 1 + (a' + b)` from associativity of plus.

2.4.5 Program Verification with Dependent Types and Type Families

Finally, applying both dependent types and type families, one can verify the operation of concatenating two lists. The operation `concat` can be recursively implemented and verified in Dependent Haskell as:

```
concat :: Vec m a -> Vec n a -> Vec (Plus m n) a
concat Nil lst2      = lst2
concat (x :> xs) lst2 = x :> (concat xs lst2)
```

The implementation is similar as before, applying pattern matching on the first parameter and recurse on the smaller list. After concatenating two lists, the resulting list is expected to have length the sum of the lengths of the two original lists. This property is then specified in the dependent type `Vec (Plus m n) a` of the result list, using the `Plus` type family.

Then, we discuss the verification in detail for the following two cases. Firstly we want to show that concatenating an empty list with any other list `lst2` results in a list of length the sum of zero and the length of `lst2`.

```
concat Nil lst2 = lst2
```

This is immediate from the definition of the `Plus` type family, as `Plus Zero n = n`. Thus, the result list `lst2` has type `Vec n a = Vec (Plus Zero n) a`.

Secondly, we want to show that concatenating any nonempty list `x :> xs` with another list `lst2` results in a list of length the sum of the lengths of the two input lists.

```
concat (x :> xs) lst2 = x :> (concat xs lst2)
```

From pattern matching, we know that the first input list `x :> xs` has type `Vec m a` and the second input list `lst2` has type `Vec n a`. Let the tail of the first input list, i.e. `xs` to have length `m'`. Then we know that `m = m' + 1` and so `x :> xs` has type `Vec m a = Vec (Succ m') a`. Given the type signature of `concat`, we then want to check if the result list `x :> (concat xs lst2)` has type `Vec (Plus (Succ m') n) a`. From the inductive hypothesis, we can show that `concat xs lst2` has type `Vec (Plus m' n) a`. It follows that, from the definition of the cons operator, the list `x :> (concat xs lst2)` has type `Vec (Succ (Plus m' n) a`. Finally, applying the second equation in the definition of the `Plus` type family on `Succ`, we can conclude that the result list has type `Vec (Succ (Plus m' n) a = Vec (Plus (Succ m') n) a`, as we expected.

2.5 F*

F*, pronounced FStar, is another functional programming language aimed at program verification [15]. It is developed at Microsoft Research, MSR-Inria and Inria and follows a similar syntax to the languages in the ML family [15]. In 2016, Swamy et al. introduced the current, completely redesigned version of F*. The new F* is a general-purpose, verification-oriented and effectful programming language. F* is said to be general-purpose as it is expressive enough to support programming in various practical domains. F* is also verification-oriented as it applies formal verification techniques to prove the correctness

of programs written in it. Finally, F* is effectful, for it allows programs with side effects, such as I/O, exceptions, stateful processing, or programs that potentially diverge.

2.5.1 Introduction to F*: Syntax, Type and Effect

F* shares most concepts with Haskell. For example, F* is also a strongly typed functional programming language. Syntactically, there are only small differences between F*'s syntax and that of Dependent Haskell. Recall the `double` function introduced in section 2.1 that doubles every element in the list. Below is an example of F*'s syntax:

```
val double: list int -> Tot (list int)
let rec double l = match l with
  | []      -> []
  | x :: xs -> (x * 2) :: double xs
```

In F*, programmers also specify the function signature before the actual implementation. All function signatures start with the keyword `val` and the type signature is introduced after a single colon instead of a double colon in Haskell. Also, all function implementations start with the keyword `let` and if the function is recursively called, the keyword `rec` will be specified between `let` and the function name. In the `double` example above, the function takes in a list of integers and returns another list of integers where each element is doubled.

It is worth mentioning that the return type in the function signature is not just the return type `list int` but a effect-and-type pair `Tot (list int)`. The prefix `Tot` is called an effect. Any function with a return type of `Tot typ` is unconditionally pure, guaranteed to evaluate to a term of type `typ` without any possibility to enter an infinite loop. This `Tot` effect specifies the function `double` to be total, or unconditionally pure. It is a special derived form of the `Pure` effect, categorizing all pure functions that always terminate without modifying any program state.

In general, a practical program does not just return a value output purely from operating on its inputs, but instead might run forever without termination, modify other states beyond its scope, or interact with its calling function or the outside world through streams and sockets. All of these operations are examples of effects. Here we briefly introduce the concept of a F* effect, which will be discussed in more detail in section 2.5.4.

For example, the effect `Pure` on total functions is F*'s logical core [15]. In a proof system like F*, a function is sound if it is correct as long as it gets accepted by the type checker. A function type checks if it complies with all rules specified in F*'s type system. With this in mind, the soundness of every pure function is determined by the required termination checks performed by the F* compiler. Thus, to emphasize, termination checking is crucial to F*.

On the other hand, different impure functions are annotated with corresponding effects.

For example, the `Dv` effect, a derived form of the general `Div` effect, is used to categorize all potentially nonterminating functions [15].

2.5.2 Simple Inductive Types

Similar to Dependent Haskell, F* allows creating new types using the `type` keyword. We show the definition of the standard library function `list` as follows:

```
type list (a:Type): Type =
  | Nil  : list a
  | Cons : a -> list a -> list a
```

The list definition in F* is similar to a `vec` GADT definition in Dependent Haskell. There are several new syntax that is worth mentioning in the above `list` definition in F*. Firstly, datatype in F* is defined in all lower case. Secondly, one applies the type parameter `(a:Type)` before colon instead of as the first type parameter after the colon. The type `a` is still considered the first type parameter of the return type, as can be seen in the constructed type `list a` of the empty list constructor `Nil`. The reason to declare `(a:Type)` before colon is to bring the generic type `a` in scope for all type definitions below. Thirdly, notice that the equality sign `=` is used instead of `where` to start the actual type definition and also notice that both type constructors begin with a capital letter. The definition of each constructor is preceded by the `|` symbol and the constructor types follow after a single colon. Finally, F* provides two syntactic sugar for the list constructors where `[]` represents `Nil` and `hd :: tl` represents `Cons hd tl`.

With the `list` definition, the `length` function can be easily programmed to return the length of any input list. Specifically, one applies pattern matching on the input list `l` with syntax `match l with` and uses the `|` symbol to separate cases. The `length` function defined on `list` is widely used in refinement logic to specify conditions for proving lemmas.

```
val length: #a:Type -> list a -> Tot nat
let rec length l = match l with
  | []      -> 0
  | _ :: tl -> 1 + length tl
```

The `_` symbol is commonly known as a wildcard, representing the value we do not care about when performing pattern matching. In the first pattern matching case, one defines the length of an empty list, `Nil`, to be `0`. In the second pattern matching case, one recursively defines the length of a list to be one more than the length of the sublist starting from the second element.

2.5.3 Intrinsic and Extrinsic Style of Verification

F* in general supports two approaches for verifying programs. Programmers can prove properties either by enriching the types of a function or by writing a separate lemma. These two approaches are often called the intrinsic and extrinsic style of verification.

2.5.3.1 Proving List Concatenation Intrinsically through Type Enrichment

Consider again the `concat` example as introduced in section 2.4.5. To concatenate two lists, one wants to prove the property that the length of the resulting list is equal to the sum of the lengths of the two input lists. Here is the definition of `concat` in F*:

```
let rec concat lst1 lst2 = match lst1 with
  | []       -> lst2
  | hd :: tl -> hd :: (concat tl lst2)
```

Type Enrichment through Dependent Types

Similar to what we have discussed in detail on Dependent Haskell's verification approach, F* supports verifying the property by indexing on the length of both input list and output list using dependent types. Firstly we introduce that definition of the `vec` GADT.

```
type vec (a:Type): nat -> Type =
   | Nil  : vec a 0
   | Cons : a -> nat -> vec a n -> vec a (n + 1)
```

Finally, using the defined `vec` datatype, the verified `concat` function is very similar to Dependent Haskell.

```
val concat : #a:Type -> #n:nat -> #m:nat -> vec a n -> vec a m ->
  Tot (vec a (n + m))
```

We want to verify that concatenating two lists, each of length `n` and `m`, will result in a list of length `n + m`. The dependent type encoding is the same as that in Dependent Haskell. There are, though, two syntactic differences in the F* version. Firstly, notice that in the above type, three extra arguments are introduced in the type definition, all are preceded by the pound sign `#`. The first argument `#a:Type` brings the generic type `a` in scope to the `concat` function. The second and third argument `#n:nat` and `#m:nat` is responsible to bring the two input lengths in scope. When preceded by the `#` symbol, the types are turned into implicit arguments. Implicit arguments tell the F* type system to try inferring the type automatically instead of always requiring programmers to provide them manually in the recursive calls. It is a kind of type inference that ease the complexity of writing code in a strictly typed environment.

Secondly, notice that instead of defining and using type family as in Dependent Haskell, we can directly apply the plus operator to specify the length of the result list. Addition, represented by the symbol `+`, is a total function defined in the library that is guaranteed to terminate. Hence F*'s type system can directly bring `+` in logic and use it in the `vec` GADT.

Type Enrichment through Refinement Types

Recall that F* supports both dependent types and refinement types. Therefore, going back to the weak typing using `list`, the same property can be verified by giving `concat` the following type:

```
val concat: #a:Type -> lst1:list a -> lst2:list a ->
  Tot (l:list a{length l = length lst1 + length lst2})
```

Notice how concise it is to enrich the function type using refinement types. All it is required additionally is the intuitive formula of the property of interest, `length l = length lst1 + length lst2`.

2.5.3.2 Proving List Concatenation Extrinsically through Lemmas

In addition to the intrinsic style of verification, F* also supports extrinsic verification through lemmas. Consider the following type signature given to the `concat` function,

```
val concat: #a:Type -> lst1:list a -> lst2:list a -> Tot (list a)
```

With the predefined `length` function, one can prove the same property as in section 2.5.3.1 using a lemma as shown below:

```
val concat_pf: #a:Type -> lst1:list a -> lst2:list a
            -> Lemma (requires True)
                     (ensures (length (concat lst1 lst2) = length lst1 + length
                        lst2)))
let rec concat_pf lst1 lst2 = match lst1 with
    | Nil      -> ()
    | hd :: tl -> concat_pf tl lst2
```

Similar to the plus operator introduced in the last section, `concat` is a total function proven to terminate. Hence, it can be directly lifted wholesale to the logic level and be reasoned about in the postcondition following the keyword `ensures`. In the `concat_pf` lemma, the property is valid in all conditions, so the precondition following the keyword `requires` is simply defined as `True`. With both precondition and postcondition specified, one can prove the lemma recursively. The `()` symbol introduced here is equivalent to `Refl` in Dependent Haskell, short for reflexivity, and denotes the satisfaction of the property.

Finally, the proof on nonempty list follows directly from inductive hypothesis.

2.5.4 Proof Automation

Various research has been done on proof automation to reduce the burden of manual proofs. Automated theorem proving is a subfield of mathematical logic that proves mathematical theorems through computer programs. Satisfiability Modulo Theories (SMT) solvers provide a good foundation for highly automated programs [12].

Satisfiability Modulo Theories (SMT) problems are a set of decision problems that generalize boolean satisfiability (SAT) with arithmetic, fixed-sized bit-vectors, arrays, and other related first-order theories. Z3 is an efficient SMT Solver introduced by Microsoft Research that is now widely adopted in program verifications [6]. Dependent Haskell does not support proof automation at any level, but the F* language achieves semi-automation through a combination of SMT solving using Z3 and user-provided lemmas.

Without automation, all proofs in Dependent Haskell are verified by the GHC compiler. Whereas, theorems in F* can be verified with help from the Z3 SMT solver. For example, in `concat_pf` above, F* can prove the postcondition `length (concat lst1 lst2) = length lst1 + length lst2)` automatically using Z3.

2.5.5 Termination Checking

As we are mostly interested in how F* handles potentially divergent functions, we will focus on the `Tot` and `Dv` effect and introduce F*'s approach towards termination checking.

F* by default requires termination checking for every function, since divergent functions may generate unsound verification conditions (VCs) and eventually end up proving some erroneous theorems. F* introduces a new termination criterion based on a well-founded partial order over all terms. A partial order is a binary relation on a set that is reflexive, antisymmetric and transitive. For example, the `<=` ordering on integers is a partial order, since for every integer `a`, `b` and `c`, it follows that `a <= a` (reflexive), `a <= b` and `b >= a` (antisymmetric), and finally if `a <= b` and `b <= c` then `a <= c` (transitive). F* insists each iteration of a recursive function be applied on a strictly smaller domain, such that the parameter of the function is guaranteed to decrease in the defined partial order during each execution. For example, consider the total function `length` introduced in section 2.5.2. In the second definition where we pattern match on a nonempty list, i.e. `_ :: tl -> 1 + length tl`, the function is recursively called on the tail of the input list `tl`. Since the tail of the list has one less element than the original list `l = _ :: tl`, F* compiler completes termination checking and accepts this definition.

Once a function is prove total, one can specify it using the `Tot` effect, notifying the F* effect system that the function is guaranteed to terminate. If, instead, one can not find any termination metric or the function diverges, F* provides the `Dv` effect, i.e. divergent, to categorize all these potentially divergent functions. Suppose that we define our `double` function as follows:

```
let rec double l = double l
```

This new definition is obviously not going to terminate as it keeps calling itself and loops forever. In this case, we have to mark the function return type with the `Dv` effect, as in `val double: list int -> Dv (list int)`. The `Dv` effect is introduced to categorize all computations that may not terminate. Specifically, it is used for intentionally divergent functions, as in the `double` example above, and others that require too much effort to prove termination. Once a function is marked as `Dv`, F* will turn off all termination checking on that function.

3 Approach and Novelty

To gain a better understanding of the language design decisions and verification techniques in Dependent Haskell and F*, we first studied the literature on each language. In general, our assessments of Dependent Haskell and F* are based on implementing and proving a wide spectrum of algorithms in both languages. These programs are written so that we can have a deeper experience with both languages and explore their features more thoroughly.

We start from implementing the datatype of a length-indexed vector in both Dependent Haskell and F*. Then we verify various functions that support types of finite length vectors from Haskell's `Data.List` module, including `length`, `head`, `tail`, `init`, `last`, `snoc`, `reverse`, `and`, `or`, `null`, etc. We then focus on the verification for various sorting algorithms, such as insertion sort, mergesort and quicksort. Finally, we examine the capability of proving divergent functions through three examples. The first example is a proof of the Peano division property `(m * n) / n = n` for the peano naturals `m` and `n` using the divergent definition of division with zero divisors. The second proof is the equivalence of multi-step and big-step evaluators in simply typed lambda calculus. Finally, we try to verify some obviously invalid proofs in each language and investigate how non-termination might interfere with normal type checking process at compile time.

We expect to structure the comparison mainly focusing on two aspects: user interface and approaches towards potentially non-terminating functions. As far as we know, we are the first researchers to directly compare the two languages Dependent Haskell and F*.

User interface refers to the interaction between programmers and programs, specifically programming simplicity when implementing programs. We are interested in examining how these two language designs differ in respect to the ease of use measured by syntactic support and library and documentation resources. As both languages are verification oriented, in addition to general programming syntax, we will also include a comparison on proof complexities for each verification technique. We believe that a good programming language should provide a clear, concise and user-friendly interface with relatively detailed library and documentation support.

The approaches towards potentially non-terminating functions refer to the two ways Dependent Haskell and F* handle functions that may not terminate. This comparison is measured in two criteria: the level of type system reliability and how verification is supported in the divergent settings. A type system is said to be strongly reliable if every property that passes compile time type check is logically valid. Both type systems in Dependent Haskell and F* are strongly reliable when we restrict the range of functions to those that are provably terminating, called total functions. However, when we extend the system to consider potentially non-terminating functions, i.e. those functions that diverge or whose termination is hard to prove, Dependent Haskell is only weakly reliable. The reason is that it type checks some verifications on logically erroneous statements and further requires run time checking to ensure program correctness. On the other hand, F* is designed to be always strongly reliable, since one must ensure the termination of all extrinsic lemmas and the correctness of compile-time verification follows. In regards to their support in the divergent settings, we are interested in whether properties on some potentially divergent functions can still be verified. Without compile time termination checking, Dependent Haskell supports theorem proving on any divergent function not much different from any other proof. However it is unknown whether F*'s type system support proving theorems on potential divergent functions of the `Dv` effect.

4 Current Results

As summarized in Section 3, we have implemented various functions that support types of finite length vectors from Haskell's `Data.List` module, three machine-checked sorting algorithms, two proofs on properties from potentially divergent functions, and finally some erroneous proofs to test the reliability of each type system.

4.1 User Interface: General Language Design and Support

In terms of user interface, we mainly focus on comparing three aspects of Dependent Haskell and F*. In Section 4.1.1, we discuss the general language design and compare the syntactic support in each language. Section 4.1.2 focuses on library and documentation resources. Finally, we present the Insertion Sort algorithm as an example to analyze the verification techniques supported by each language.

4.1.1 Syntactic Support

Throughout our analysis, we conclude that Dependent Haskell is syntactically inelegant while F* has a fairly friendly user interface.

4.1.1.1 Dependent Haskell

Dependent Haskell enforces phase separation between the term and type level. Thus, to apply a function defined in the term level to a dependent type in the type level, programmers are often expected to repeat implementations of the same logic.

The first type of duplication is in data type definitions. Dependent Haskell enables dependently typed programming through the use of singletons. Singleton is a special type, also known as a singleton type, that bridges the gap between run-time values and compile-time types. Each singleton type is indexed by a type variable of the promoted kind (recall from section 2.4.1 that kind is a classification of types). For example, `SNat :: Nat -> Type`, a GADT indexed by a type of kind `Nat` and returns a new type `Type`(or `*`), is a singleton type for the `Nat` datatype on natural numbers. A singleton type is a type with exactly one value, where each type variable indexing the singleton type is uniquely mapped to the term of that type. Hence there is an isomorphism between one singleton type and its associated values, ensuring that term-level computation and type-level computation always go hand in hand [8].

Take the natural numbers as an example, there are two different definitions – the original datatype `Nat` and the derived singleton type `SNat`. Prior to the introduction of the `singletons` library, programmers need to manually implement the singleton type, even though the definition of a singleton type is a straightforward extension to that of the original datatype.

The `singletons` library introduced by Eisenberg and Weirich aims to remove the code duplication. Using Template Haskell, the library can automatically generate the definitions for singleton types [8]. However, the programming overhead still remains, as the library only frees programmers from the tedious singleton definitions but does not hide the nature of the internal singletons encoding. Programmers are still expected to have a deep understanding of the phase separation and the singleton types, and to take full charges of the function implementations using the library-generated definitions.

The second type of duplication comes from function definitions. Functions such as adding two natural numbers, need to be specified once in the term level for the original datatypes, once in the type level using type families, and once in the term level for the singleton types. These three versions of the same addition algorithm on natural numbers add in lots of additional burdens when using the language. To free programmers from these repetitive and tedious works, the `singletons` library also attempts to automate the function definition process. Through Template Haskell, the library supports promoting term-level functions and directly reuse them in the type level [8]. It also supports enriching functions with richer types, extending the term-level definitions to directly apply on singleton types. From our experience with the library, the Template Haskell syntax is easy to pick up for programmers already familiar with verification programs. However, the actual development process is not very smooth. When proving relatively more complicated theorems, like the STLC proof introduced in section 3, we often need to refer back to the library-generated definitions to complete the verification.

In summary, the `singletons` library reduces programmer's burden to manually implement all functions at different level through the automation of singleton data type definition, and the generation of type family and enriched functions on singleton types. Yet, the library still does not resolve the syntactic problem in Dependent Haskell. The `singletons` library only helps lessen the effort users need to spend on the duplicated declarations, but it does not abstract away any detail of the actual definitions. In addition, the generated functions contain a lot of renaming for keywords, terms and types. As a result, all the generated data types and functions after turning on the `ddump_splices` compiler option are very hard to read. In turn, compare to debugging using self-defined data types and functions, proofs using the generated structures sometimes add additional complexity to parse error messages. Besides, programmers still need to fully understand how type family and singletons work to compile their programs and to complete the verifications. Therefore, even with the assistance of the `singletons` library, Dependent Haskell's syntax remains a little clumsy and is not very friendly to users new to programming verification.

4.1.1.2 F*

On the other hand, F* is syntactically concise. It supports both type-level specifications through dependent types and type-level computations through refinement types. For example, the type signature for the `concat` function introduced in section 2.4.5 can be specified using dependent type as `val concat: #a:Type -> #n:nat -> #m:nat -> vec n a -> vec m a -> vec (n + m) a`. F*'s approach towards dependent types is similar to that of Dependent Haskell, except that F* frees programmers from the repetitive type family implementation of `Plus`. Through the `T-Ret` typing rule using the idea from monadic returns (we will not go into detail for the typing rule) [15], F* can automatically enrich the types of a total function and reflect the existing term-level implementations into logic for reasoning. Specifically in the example of `concat`, the library function `(+)` is automatically reflected to the type level and is applied in the dependent type `vec (n + m) a`.

The same verification for concatenating two lists can also be specified using refinement type, as `val concat: #a:Type -> #n:nat -> #m:nat -> lst1:list a -> lst2:list a -> r:list a{length r = length lst1 + length lst2}`. Instead of indexing the length of the vector to form dependent types, we can directly specify the condition `length r = length lst1 + length lst2` as a logical predicate to the resulting type.

4.1.2 Library and Documentation Supports

Dependent Haskell has many well-documented and timely-maintained libraries as well as a lot of online resources. However, the relatively new F* language is still in the process of building up libraries and adding documentation.

Dependent Haskell, extending Haskell with dependent types, has been officially implemented in GHC 8.0 and is now a part of Haskell. As Haskell is a mature, industrial-

strength programming language, Dependent Haskell inherits all its resources. Specifically, Haskell provides a central package archive called Hackage where packages are introduced, maintained, and supported. In addition, Hoogle is a holistic search engine for Haskell APIs which provides advanced queries through function names such as `sum`, and even through approximate type signatures such as `int -> int -> int`. As Haskell becomes more popular in the real world, there emerges a plethora of online tutorials and textbooks with detailed explanations of syntax, concepts and algorithms. Since the introduction of Dependent Haskell, tutorials, such as schoolofhaskell, have gradually incorporated Dependent Haskell into their documentations. Resources for Dependent Haskell are also available on Richard Eisenberg's Github repository for his dissertation (https://github.com/goldfirere/thesis).

However, as a completely redesigned language that is relatively new, F* does not have many resources yet. Besides various conference publications, F* has an official website organizing all of its resources (https://www.fstar-lang.org). Most of the documentations are through its interactive tutorial (https://www.fstar-lang.org/tutorial/) and examples in the official FStar Github repository (https://github.com/FStarLang/FStar). In addition, source code for each library function can be found in the `ulib` folder of its Github repository (https://github.com/FStarLang/FStar/tree/master/ulib). There are a lot of functions implemented, but they are mostly shown as the original implementation without much additional documentation.

4.1.3 Proof Complexity

Regarding verification techniques, F* provides a very intuitive user interface that engages users in the process of writing proofs. In addition, the SMT solver provides great automation that frees users from many tedious proofs. On the other hand, theorem proving in Dependent Haskell has a steep learning curve and is already laborious for some basic sorting algorithms.

The current Dependent Haskell as implemented in GHC 8.0 does not support any proof interaction or automation. Verification in Dependent Haskell is expressive, but could get really tedious even for relatively simple sorting algorithms, like insertion sort, mergesort or quicksort. F*, on the other hand, provides a flexible combination of SMT-based proof automation and manually constructed proofs. In addition, F* can use SMT solver, such as Z3, to match SMT patterns provided in the manually provided lemmas to assist with user-constructed proofs. F* supports theorem proving through either enriching function types (intrinsic style) or by writing separate lemmas on properties (extrinsic style) [15]. It categorizes effects into `Tot`, `Dv`, etc, requires explicit annotation of the effect in type signatures and enforces termination checking on all total functions.

In this section, we will focus on comparing how Dependent Haskell and F* vary in their approaches to prove properties on insertion sort. It is a simple sorting algorithm that builds the resulting sorted list by inserting each input element to the correct position one at a time. The insertion sort algorithm is chosen in this comparison for the following

three reasons. Firstly, it is a widely-used algorithm that is easy to understand for most computer science majors. Secondly, the correctness of the sorting algorithm is well-defined and important to be verified. Thirdly, it features an algorithm with nontrivial properties that are hard to prove intrinsically. Specifically, to prove that a sorting algorithm is correctly implemented, we need to check both contents and ordering of the resulting list. The contents of the output list should remain the same as the input list. In other words, the output list is a permutation of the input list. Also the resulting list must be verified to be sorted, meaning that it has a monotonically non-decreasing order.

To ensure the fairness of the comparison, verifications in both languages are based on the same insertion sort definition. Also without loss of generality, we abstract out the details of generic types and focus on discussing verification on a list of natural numbers in this paper. The base algorithm for insertionSort is shown below in Haskell syntax and we will discuss it in detail when talking about the proofs.

```
insert_sorted :: Nat -> [Nat] -> [Nat]
insert_sorted n []   = [n]
insert_sorted n (h:t) = if n <= h then (n:h:t) else h:(insert_sorted n t)

insertionSort :: [Nat] -> [Nat]
insertionSort []    = []
insertionSort (h:t) = insert_sorted h (insertionSort t)
```

In the following sections, we present the proof in both languages and will discuss the comparison at the end. Section 4.1.3.1 features the implementation in Dependent Haskell and section 4.1.3.2 features that in F*.

4.1.3.1 Dependent Haskell

Our verification of insertion sort in Dependent Haskell defines two GADTs, `PermutationProof` and `NonDecProof`, each encoding one property that we want to show. The actual proof, as shown below takes in a singleton list and returns a tuple of proofs. By using a singleton list `SList lst`, Dependent Haskell duplicates the term level list `lst` to the type level and allows the list to be reasoned in the type signature.

```
insertionSort_pf :: SList lst -> (PermutationProof lst (InsertionSort lst),
                                  NonDecProof (InsertionSort lst))
insertionSort_pf slst = (permutation_pf slst,
                         nondec_pf slst)
```

Firstly, we discuss the proof that the list after insertion sort is a permutation of the input list, i.e. the lemma `permutation_pf` that takes in a singleton list and returns the defined `PermutationProof`.

To begin with, our permutation proof requires the definition of the `PermutationProof` GADT:

```
-- A proof term asserting that lst1 and lst2 are permutations of each other
data PermutationProof (lst1 :: [k]) (lst2 :: [k]) where
  PermId  :: PermutationProof l l
  PermIns :: InsertionProof x lst2 lst2' -> PermutationProof lst1 lst2 ->
             PermutationProof (x ': lst1) lst2'
```

The `PermutationProof` encodes a proof that ensures that the original list `lst1` and the list `lst2` after insertion sort are permutations. We say that two lists are permutations of each other if they contain the same set of elements. The proof contains two cases. The trivial case `PermId` states that the same two lists are permutations. The inductive case `PermIns` states that adding the same element to both lists that originally contain the same set of elements will result in two lists that remain a permutation of one another. InsertionProof is a proof term asserting that `lst2'` is the list produced when `x` is inserted anywhere into the list `lst2`.

Next, given the proof term `PermutationProof` defined above, we can implement the lemma `permutation_pf` as follows:

```
-- A lemma that states that the result of an insertion sort is a permutation
-- of its input
permutation_pf :: SList lst -> PermutationProof lst (InsertionSort lst)
permutation_pf lst = case lst of
  SNil -> PermId
  SCons h t ->
    case insert_permutation_pf h (sInsertionSort t) of
      insertionPf -> PermIns insertionPf (permutation_pf t)
```

Lemma `permutation_pf` takes in a singleton list and returns the proof ensuring that the result of an insertion sort is a permutation of its input. Similar to all lemmas introduced before, this lemma also starts with pattern matching on the input list. If the input list is empty, i.e. in the first case `SNil`, then we can immediately conclude that two empty lists are permutations by `PermId`. Otherwise, when we have a nonempty input list of structure `h:t`, i.e. in the second case `SCons h t`, we need to apply another lemma to complete the proof. We first verify through the `insert_permutation_pf` lemma that inserting `h` into the insertion sort of `t` returns a correct new list with `h` inserted. Then, we recursively check if the tail `t` is a permutation of the insertionsort of `t`. Finally, we can directly construct the resulting proof term using the `PermIns` value constructor as defined. To check if inserting an element works correctly, we implement the lemma `insert_permutation_pf :: SNat n -> SList lst -> InsertionProof n lst (Insert n lst)`. The lemma takes in a term `n` of singleton type `SNat` and a singleton list `lst`, and returns a proof ensuring that inserting the element `n` into the list `lst` produces the same list as the result of calling `Insert n lst`.

Similarly, but with more steps, we can prove that the resulting list is sorted with a nondecreasing order. As a result, the insertion sort implementation introduced in section 4.1.3 meets the specification in full — the resulting list is both sorted and a permutation of the input list. Finally, since our Dependent Haskell proof on insertion sort type checks,

we can conclude that the insertion sort implementation is verified to be correct.

4.1.3.2 F*

Since F* supports refinement types, we do not have to define our own proof terms as in the proof in Dependent Haskell (though proving using dependent types is also supported). Instead, we can directly prove the property intrinsically by providing refinement conditions for the `insert_sorted` and `insertionSort` functions.

The function signature of `insertion_sort` with refinement types is shown below in F*'s syntax:

```
val insertion_sort : lst1:list nat
    -> Tot (lst2:list nat{(forall k. is_elem k lst2 <==> is_elem k lst1) /\
                          (sorted lst2)})
```

The above `insertion_sort` function takes in a list of natural numbers and returns another list of natural numbers. Recall that there are two properties that we want to show in an insertion sort, i.e. the contents and the ordering of the resulting list. In the refinement predicate, we prove the permutation lemma by stating that `forall k. is_elem k lst2 <==> is_elem k lst1` and we specify that the resulting list is sorted through `sorted lst2`.

Regarding the Permutation Predicate Notice that, unlike the permutation proof in Dependent Haskell, our definition of permutation above only addresses the case when both lists do not have duplicated elements. We can, however, update the refinement predicate to `forall k. count_elem k lst2 = count_elem k lst1` and provide the definition of `count_elem` as follows:

```
val count_elem: nat -> list nat -> Tot nat
let rec count_elem x l = match l with
  | []       -> 0
  | hd :: tl -> if hd = x then 1 + count_elem x tl else count_elem x tl
```

Due to time limits, we do not finish the modified implementation, but we do not expect the complete verification to be much different than the one presented here.

Our `insertion_sort` has a fairly complicated refinement predicate —— `(forall k. is_elem k lst2 <==> is_elem k lst1)` $\wedge$ `(sorted lst2)`. It introduces a lot of new syntax, and we will discuss them one at a time:

1. The `forall k.`quantifier is the universal quantifier and the `<==>` symbol is a propositional connective for bidirectional implication. So the first part of our predicate states that every element in the resulting list `lst2` is an element in the input list `lst1` and every element in the input list `lst1` is also an element in the resulting list `lst2`.

2. F* provides the propositional connective $\wedge$ for conjunction, which connects two types similar to how `&&` connects two booleans. So the above refinement predicate specifies that the resulting list contains exactly the same unique elements as the input list and is sorted in a non-decreasing order.

3. Both `is_elem` and `sorted` are helper functions we provide to help specify the theorem. The function `is_elem` checks whether a natural number is an element of the list of natural numbers. The function `sorted` takes in a list and returns a boolean indicating whether it is sorted. As we will refer to the definition of `sorted` in our discussion follows, we present its definition below:

```
val sorted: list nat -> Tot bool
let rec sorted l = match l with
   -- an empty list is sorted
    | []           -> true
    -- a single element list is sorted
    | [x]          -> true
    -- a list with more than one element is sorted if the first two elements
        are sorted and the tail of the list is recursively checked to be sorted
    | x :: y :: xs -> (x <= y) && (sorted (y :: xs))
```

We just briefly discuss the implementation in code comments, so we only expect our readers to get familiar with this definition when it gets mentioned in the following proofs.

Verification on Insertion Sort

Following the same algorithm as shown in the beginning of section 4.1.3, we define the `insertion_sort` function in F* as follows:

```
let rec insertion_sort lst1 = match lst1 with
    | []       -> []
    | hd :: tl -> insert_sorted hd (insertion_sort tl)
```

Intrinsic Proof Style With our defined refinement predicates, the proof for insertion sort is mostly carried out intrinsically. In the intrinsic style of proving, every call of the function carries all properties specified in the refinement type and is intrinsically proven about the function. As such, F* relieves programmers from writing separate lemmas and allows them to focus on the specifications in refinement predicate instead.

To illustrate this intuitive proof writing process, we will go through the verification of `insertion_sort` in detail. Similar to most verifications we have introduced before, the proof starts by pattern matching on the input list. Let us separately discuss the following two cases:

Firstly, consider the base case, when we have an empty input list. The implementation of this case follows:

```
[] -> []
```

Sorted Proof By definition of the `sorted` lemma, we know that `sorted [] = true`. But both the input list and the resulting list is an empty list. Hence it is immediate that `sorted lst1 sorted [] = sorted lst2`.

Permutation Proof We want to show that the input empty list has the same element as the resulting list, which is also an empty list. It follows immediately from our definition of `is_elem` that `forall k. is_elem k lst1 = is_elem k [] = false = is_elem k [] = is_elem k lst2`.

Both proofs directly follow from the definition of our lemmas and can be carried out automatically in F*. Then we discuss the case, as shown below, when the input list contains at least one element.

```
hd :: tl -> insert_sorted hd (insertion_sort tl)
```

We want to prove that inserting the head of the input list into its tail satisfying both sorted and permutation specifications will result in an output list as desired. Specifically, the refinement predicate `(forall k. is_elem k lst2 <==> is_elem k lst1) /\ (sorted lst2)` ensures that the output list is both a permutation of the input list `hd :: tl` and is sorted with a non-decreasing order.

Both sorted and permutation properties can be directly derived from the refinement predicate of the `insert_sorted` function. Below we show its function signature:

```
val insert_sorted : x:nat -> lst1:list nat{sorted lst1} -> Tot (lst2:list
    nat{sorted lst2 /\ (forall k. elem k lst2 <==> k == x \/ elem k lst1)})
```

Recall that `<==>` is the propositional connective for bidirectional implication. So the refinement predicate of `insert_sorted` ensures that every natural number is an element of the output list if and only if it is the newly inserted element or it is an element of the input list.

Below is the F* implementation of `insert_sorted` following the same algorithm introduced in section 4.1.3.

```
let rec insert_sorted x lst1 = match lst1 with
  | []       -> [x]
  | hd :: tl ->
     if x <= hd then
       x :: lst1
     else
       hd :: (insert_sorted x tl)
```

We will not go over each step of the verification as most cases can be proved directly following the inductive hypothesis or the definition of `is_elem` and `sorted`. The inductive hypothesis states that inserting element `x` into the tail of the input list results in a sorted list with exactly one more element `x` than the tail list.

The only interesting part of this verification is the sorted proof when we have a non-empty input list and the inserted element `x` is greater than the list head. We show its implementation separately as follows:

```
  | hd :: tl ->
     if x <= hd then
       x :: lst1
     else
       hd :: (insert_sorted x tl)
```

Following the implementation, we want to show that the list `hd :: (insert_sorted x tl)` is sorted. Notice that since `x > hd`, this list has more than one element. Let the head of the list `hd :: (insert_sorted x tl)` be `y`. By definition of our `sorted` lemma, it suffices to show that `hd <= y` and `sorted (insert_sorted x tl)`.

By inductive hypothesis, inserting element `x` into the tail of the input list results in a sorted list, so we can conclude immediately that `insert x tl` is sorted.

We then focus on proving `hd <= y`. To finish the proof, we need to introduce a new lemma to assist with the Z3 SMT solver and we present its function signature below:

```
val smaller_hd_lemma: m:nat
                   -> n:nat
                   -> lst:list nat
                   -> Lemma (requires (sorted (m::lst) /\ is_elem n lst))
                            (ensures (m <= n))
                      [SMTPat (sorted (m::lst)); SMTPat (is_elem n lst)]
```

The `smaller_hd_lemma` lemma states that given any sorted list, the head is always less than or equal to any element in the tail of the list. Recall that the preconditions follow after the `requires` keyword and the postconditions follow after the `ensures` keyword. So our extrinsic verification condition ensures that if the list `m :: lst` is sorted and `n` is an element of its tail `lst`, then we have `m <= n`.

SMT Lemmas to Bridge the Gap between Intrinsic and Extrinsic Proofs

The property stated in `smaller_hd_lemma` is beyond the scope of the Z3 solver to verify automatically. So we have to manually provide the above lemma to assist with it. However, we want to avoid polluting our `insert_sorted` implementation with calls to this supporting lemma. Instead, we would like to extend the intrinsic proving style to directly associate the theorems proven from the lemma to the function `func` we are trying to prove. As such, every application of the function `func` can implicitly carry these proven theorems. SMT Lemma is such a mechanism that F* provides to improve the experience of extrinsic lemma proving [2].

Notice that the type of `smaller_hd_lemma` above introduces one new syntax, a list of `SMTPat`s. The `SMTPat` keyword is short for an SMT pattern. It instructs F* and the Z3 solver to pattern match on the specific terms provided following the keyword such that every occurrence of the specified pattern will be associated with the properties proven from the lemma.

Specifically, our `smaller_hd_lemma` lemma pattern matches on the two terms from our precondition, `sorted (m::lst)` and `is_elem n lst`. Whenever Z3 finds a matched pattern, it will implicitly applies the `smaller_hd_lemma` lemma and uses the associated property in the postcondition to complete the proof.

With the newly introduced lemma, we then finish our discussion on the proof `hd <= y`. Recall that we denote the head of the list `insert_sorted x tl` as `y`. Let its tail be `rest`. Then we can denote this list `insert_sorted x tl` equivalently as `y :: rest`. If `y == x`, then we can immediately conclude that `hd <= y` since we are under the case `hd < x`.

The proof `hd <= y` when `y` $\neq$ `x` is carried out automatically by the Z3 SMT solver, implicitly using the helper lemma `sorted_smaller_hd` as defined above. Since `y` $\neq$ `x`, we can conclude that `y` is an element of `tl`. Thus, given the preconditions that `lst1@(hd :: tl)` is sorted and `is_elem y tl`, we conclude from the postcondition of the helper lemma that `hd <= y`.

4.1.3.3 Summary

From a thorough comparison of the insertion sort proofs in both Dependent Haskell and F*, we conclude that F* is comparably easy and relatively intuitive to program with. The complete verification in F* only takes less than 30 lines of code, but the same verification in Dependent Haskell takes almost 100 lines of code.

Below we summarize the key features of F* that contribute to its user-friendly interface for verification:

Refinement Types With the support of refinement types, F* saves programmers from the burden of implementing proof terms. Instead, it integrates the proofs almost seamlessly into the original implementations. Start from a function, such as the `insertionSort` above, we can directly encode the logic in the refinement predicate of its

function signature and provide helper lemmas to complete the verification when necessary.

Proof Automation with SMT Solver Recall from section 2.5.4 that F* achieves semi-automation through a combination of SMT solving using the Z3 solver and user-provided lemmas. Once we encode the theorems using refinement types, F* will generate verification conditions (VC) based on them and try to discharge the proofs automatically through the Z3 SMT solver. For example, in the sorted proof on `insert`, the transitivity of the `<=` operator is automatically proved by Z3. When there is a gap in the verification beyond the ability of Z3, for example in the second case of `insert`, users can provide a lemma to fill in the gap. F* provides the `SMTPat` keyword that supports pattern matching on logical predicates, so Z3 can automatically carry out the provided helper lemma to complete the proof.

Intuitive programming with Syntactic Sugar Finally, as discussed in section 2.2 that a refinement type, of the form `x:t{phi(x)}`, provides an additional logical predicates on the term of type `t` [2]. Internally, the predicate `phi(x)` is a type dependent on the term that will be transformed into VCs by the F* type system. F* provides a lot of propositional connectives, such as `==`, $\wedge$, $\vee$, $\sim$, `<==`, `==>`, `<==>` etc, to hide the details of type construction. These syntactic sugar allows programmers to use boolean functions directly in the predicate instead of messing with types, and further improves the experience of theorem proving in F*.

On the other hand, Dependent Haskell requires encoding for all proof terms using dependent types. Programmers also need to provide every single step of the proof due to a lack of automation. Programming in Dependent Haskell requires a deeper understanding of type theory and can be tedious and hard.

Yet, throughout our comparison on proving some more complicated theorems, we find that F*'s interface gets harder to use quickly, gradually approaching the complexity of Dependent Haskell. Firstly, intrinsic verification style through type enrichment is not always possible for many complicated properties. Then programmers must resort to the more verbose extrinsic style of verification using lemmas. Secondly, when proving more complicated theorems, we often go beyond the ability of the Z3 solver, so we end up writing many additional lemmas to assist with Z3. In addition, bad patterns introduced with the `SMTPat` keyword may result in unpredictable performance from the verification engine [2], but good specific patterns are harder to come up with when proving more involved properties.

In a nutshell, we believe that F* does make the relatively easy proofs easy, but the proofs on hard properties still remain hard. In general, we conclude that F* has a very good abstraction over its type system that provides an intuitive and friendly user interface for theorem proving.

4.2 Approaches towards Potentially Non-terminating Functions

In the second part of our comparison, we focus on potentially non-terminating functions. We say a function terminate, or is total, if it returns a value for all possible inputs and we say a function diverges if it gets stuck in an infinite loop. Due to the undecidability of the Halting problem, termination analysis can not be accurate. As a result, the inaccurate analysis often leads to either false positives when divergent functions are mistakenly checked to terminate, or false negatives when terminating functions are checked to diverge. Programmers generally design type systems to minimize false positives, sometimes resulting in an increase in false negatives.

Dependent Haskell and F* each takes a different approach towards handling functions that may not terminate. Dependent Haskell is a partial language whose function does not guarantee to return a value. Also, since its type safety does not depend on termination or totality checking, it does not have a termination checker [7]. F*, however, aims to be a proof assistant in which termination checking is crucial to ensure the correctness of its proofs [15]. It provides a fully semantic termination criterion and guarantees that all total functions are recursively defined on a strictly smaller domain (see section 2.5.5 for a detailed discussion on F*'s termination checking).

When verifying theorems, each of these two approaches has its strengths and limitations. Through our comparison, we conclude that Dependent Haskell is able to prove properties on potentially divergent functions. Yet it is weakly reliable, since expressions of some erroneous theorems of type equality (such as $1 :\sim: 2$) can pass type check at compile time, causing the GHC compiler to loop [7]. In other words, Dependent Haskell can not assure the correctness of some properties at compile time and additionally requires further validation at run time. On the other hand, F*'s type system is strongly reliable but is in lack of support on proving functions that may not terminate. Even though F* allows programmers to turn off termination checking on a per function basis, it has not yet support a satisfiable approach to prove properties on these functions of the `Dv` effect. Both limitations compromise the motivation of a verification-oriented programming language, but achieving both in full is a real theoretical challenge. Therefore, we want to compare and analyze trade-offs between Dependent Haskell's and F*'s approach towards non-termination.

We start from exploring the Peano division example (whose complete implementations in both languages are provided in Appendix) where we explicitly know the reason of non-termination. Peano naturals are a simple way to encode natural numbers with the `Zero` base value and the `Succ` recursive successor, each time incrementing the value by one. For example, natural number 0 is expressed as `Zero`, 1 is expressed as `Succ Zero` and 2 is expressed as `Succ (Succ Zero)` etc. In our Peano division example, we want to prove the property that given two natural numbers `m, n` with `n > 0`, the quotient of their product and n is equal to m, i.e. `(m * n) / n = m`. Recall that division is often defined with a nonzero divisor. However, we allow zero divisors in our definition of division(`/`) to examine how Dependent Haskell and F* perform verifications on potentially divergent functions.

4.2.1 Dependent Haskell

4.2.1.1 Dependent Haskell can prove properties on potentially divergent functions

Dependent Haskell does not require termination checking and the type safety of Dependent Haskell programs does not depend on the termination checker [7]. In other words, Dependent Haskell ensures no untrapped errors, such as segmentation fault, when handling nontermination. With the help from type families and singletons, Dependent Haskell can successfully verify the Peano division property.

We first define the Peano natural numbers. The definition of the `Peano` ADT is similar to that of `Nat` as introduced in section 2.4.1. In fact, our `Nat` GADT is defined following the peano axioms.

```
data Peano = Zero | Succ Peano
```

Note that the Peano natural numbers are nonnegative. So we define the division type family as follows:

```
-- recursive definition of integer division (/)
type family (m :: Peano) / (n :: Peano) :: Peano where
  Zero / _ = Zero
  m / n    = (Succ Zero) + (m - n) / n

-- (+), (-), and (*) are defined similarly
```

When dividing Peano natural number `m` by `n`, we first subtract the divisor `n` from the dividend `m`. Then we recursively divide the difference by `n` and finally increment the quotient by `1`. In order to simplify the type family implementation and focus our discussion on the verification, we make the following two simplifications. Firstly, we define the result of `Zero / Zero` to be `Zero`. Secondly, we define our division to always round up. Consider the example calculating the quotient of `3` and `2`. Unwrapping the definition of division and subtraction, we get `3 / 2 = 1 + (3 - 2) / 2 = 1 + (2 - 1) / 2 = 1 + (1 - 0) / 2 = 1 + 1 / 2 = 1 + 1 + (1 - 2) / 2 = 1 + 1 + 0 / 2 = 1 + 1 + 0 = 2`, where we simplify the peano naturals with natural numbers.

As long as divisor `n` is nonzero, we are recursively calling division on a smaller dividend, so division is guaranteed to terminate. As we want to explore properties on nonterminating functions, we deliberately give `n` the type `Peano`, including the case when `n = Zero`.

Other operations `+`, `-` and `*` are defined similarly. Since all four operations are defined in the type level using type families, they can be used in the type signature of the proof `peanoDivisionPf`.

Finally, we prove the property `(m * n) / n = m` for positive `n` in `peanoDivisionPf`.

Note that the property `(m * n) / n = m` is only valid when `n` $\neq$ `0`, so we restrain the divisor to be positive to ensure the validity of the theorem. Using the corresponding singletons for `Peano`, we can define the positive divisor as `SPeano (Succ n)`. So the property becomes `(m * (Succ n)) / (Succ n) = m`. Below is the type signature of the `peanoDivisionPf` function:

```
peanoDivisionPf :: SPeano m -> SPeano (Succ n) ->
  (m * (Succ n)) / (Succ n) :~: m
```

The actual implementation of `peanoDivisionPf` is not very important here, though interested readers can go through the complete proof in Appendix 2.1. Similar to all verification examples introduced before, the proof of theorem `(m * (Succ n))/(Succ n):~:m` is carried out inductively. We pattern match on the dividend, then unwrap the definitions of each operator, apply a list of predefined lemmas, and recursively use the inductive hypotheses until we reach the property we want to show.

Dependent Haskell does not perform compile-time termination checking. It allows programmers to define divergent functions like division with zero divisors, and tries to evaluate them in the function signature. If the theorem is valid, such as the property `(m * (Succ n))/(Succ n):~:m` we want to prove in our peano division example, then the evaluation on the divergent function will terminate with a valid proof.

4.2.1.2 Dependent Haskell is weakly reliable towards verifications on type equalities

In the Peano division example above, we have successfully verified the valid property `(m * (Succ n)) / (Succ n) = m` on our divergent division(`/`) type family. However, we then wonder how Dependent Haskell would handle the case if we try to prove an invalid theorem. So we modify the function signature of `peanoDivisionPfLoop` as follows:

```
peanoDivisionPfLoop :: SPeano m -> SPeano n -> (m * n) / n :~: m
```

Notice that we have removed the restriction on the divisor and allow it to be zero. As such, whenever we pattern match on `n` and get `SZero`, the theorem becomes `(m * Zero) / Zero :~: m`. Recall that we define the equation `Zero / Zero` to be `Zero`, so this modified theorem of type equality is invalid as long as `m` $\neq$ `Zero`. Thus, we expect any proof on this theorem with a nonzero dividend but a zero divisor to be rejected by Dependent Haskell. Below we provide one such proof that simply loops by recursively calling itself.

```
peanoDivisionPfLoop :: SPeano m -> SPeano n -> (m * n) / n :~: m
peanoDivisionPfLoop (SSucc m) SZero = peanoDivisionPfLoop (SSucc m) SZero
```

When we compile the above program, it actually passes type check at compilation. In other words, GHC seems to accept it. But when we run the program with `peanoDivisionPfLoop (SSucc SZero) SZero`, for example, it simply loops without returning any value.

Unlike many proof systems, Dependent Haskell can not assure the correctness of every proof term that is type checked at compile time. In fact, all equality proofs in Dependent Haskell must be run to determine their validity. For this reason, we call Dependent Haskell's type system weakly reliable. Generally, there are two run time outcomes for functions in Dependent Haskell that pass type check; they either return a value or diverge. When a function terminates with a value, the equality proof it returns is guaranteed to be correct and so is accepted by Dependent Haskell. On the other hand, when a function diverges, as in the `peanoDivisionPfLoop` example above, the property it carries is invalid and so is rejected at run time. In other words, erroneous proofs might pass type check and can only be discovered at run time when running them causes GHC to loop.

In a nutshell, equality proof terms themselves do not give any promise when they pass type check at compile time. In effect, assurance is only given if the proof not only passes the type check but also returns a value at execution.

Since Dependent Haskell is weakly reliable, verification in it is sometimes counterintuitive. When proving theorems with type equality, the propositional equality term `:~:` that many programmers treat as a proof is actually not a proof with any guarantee. Furthermore, as a result of the additional run-time checking, these execution-irrelevant proof terms must remain in type during execution, resulting in a worse run-time performance in Dependent Haskell.

In summary, Dependent Haskell is able to prove potentially non-terminating functions, but this capability comes with its weak reliability. As Eisenberg acknowledged, this drawback is indeed serious and the only way to get around it is by designing a suitable termination checker [7]. In the next section, we will switch our focus to F* and examine how its effect system with a well designed termination checker performs in the divergent settings.

4.2.2 F* is in lack of a satisfiable approach in the divergent settings

Consider again the Peano division example. We define the `peano` GADT in F* as follows:

```
type peano: Type =
 | Zero : peano
 | Succ : peano -> peano
```

4.2.2.1 The Lack of Support to Prove Extrinsically

To begin with, we try to prove the theorem `(m * n) / n = m` in the extrinsic style, using the following definition of `division`:

```
val division: a:peano -> b:peano -> Dv (c:peano)
```

To explore whether F* has the ability to prove properties on potentially non-terminating functions, we again deliberately define a divergent `division` function. By definition of the `peano` type, the divisor `b` could be zero. Hence this `division` function defined above may not terminate and we have to annotate its effect to be `Dv`. The function signature for our peano division proof is shown below:

```
val peanoDivPf: m:peano -> n:peano{toNat n > 0} -> Lemma (ensures (division
    (mult m n) n == m))
```

Compiling the above code results in an error. The error message states that a ghost expression is expected but the type system gets an expression with a divergent effect. Ghost is another effect predefined in F*'s type system that encapsulates computationally irrelevant code [15]. For example, `unit` is such an example that holds no information. When a function returns a `unit`, all we care about are the properties it carries. These properties can be specified in the refinement predicate of the unit value, or after the `ensures` clause using the syntactic sugar, and are verified at compile time.

The error occurs because we are trying to apply the divergent `division` function with effect `Dv` to the postcondition that expects a computationally irrelevant total expression of effect `GTot` (stands for Ghost Total). We say a function is total if it is defined for every input in its domain. By this definition, every total function is provably terminating. The above usage of `division` in the postcondition of the `peanoDivPf` lemma is rejected by the F* compiler since it cannot derive a total expression out of a function that may not terminate. As a result, even though F*'s type system can accept potentially divergent functions of the `Dv` effect, it refuses to evaluate them during compile-time type checking.

To determine whether F* is able to verify potentially divergent functions, we then focus on exploring F*'s approach to prove intrinsically through type enrichments.

4.2.2.2 An Unpractical Approach to Prove Intrinsically

A Valid Proof using the Terminating Division Function

To better understand how Peano division proof works in F*, we first start with a discussion on a valid proof encoding for the terminating division function. As we clearly know the reason of non-termination in this exploratory Peano division example, we can easily define a division function that terminates. Specifically, we provide a refinement condition to the divisor `b` to ensure that it is nonzero. The refined definition of `division_terminating` is shown below:

```
val division_terminating: a:peano -> b:peano{toNat b > 0} -> Tot (c:peano)
  (decreases (toNat a))
```

There are two concepts worth mentioning in this `division_terminating` definition. Firstly, the `toNat` function takes in a `peano` GADT and returns the corresponding natural number, i.e. `toNat Zero = 0` and `toNat (Succ Zero) = 1` etc. Secondly, recall from section 2.5.5 that F* needs to perform termination checking for all `Tot` functions. Specifically for the `division_terminating` function above, the logic is too complicated for F* to automatically deduce a metric from its built-in termination metrics. As a result, we need to explicitly provide our own metric using the `decreases` keyword. Remember from the previous section on Dependent Haskell that division is defined as `1 + (a - b) / b`. We can quickly find that given a nonzero divisor `b`, the dividend `a` is decreasing at each recursive step. Hence we provide the termination metric as `decreases (toNat a)` following the main type signature.

With the above refinement and our explicit termination metric, the division function can be proven to terminate. Hence, we can mark `division_terminating` with the `Tot` effect and our proof can then be successfully discharged with the following function signature:

```
val peanoDivPf: m:peano -> n:peano{toNat n > 0} ->
  Lemma (ensures (division_terminating (mult m n) n == m))
```

Notice that this valid proof is written in the extrinsic style, proving the property with an additional lemma.

A Case-Specific Workaround

Next, we return back to the divergent `division` definition with zero divisors and move on to discuss a Peano division proof that is successfully verified through type enrichment.

As F* does not support extrinsic proofs on non-terminating computations, we shift our focus to explore the possibility to prove intrinsically. However, the lemma `(a * b) / b = a` we want to prove is not a condition on any single function, but a combination of both multiplication and division. As such, we cannot simply refine the type of a function. Dependent types also failed here since the logic is too complicated to be encoded only using types (it might be possible but really hard in practice). Instead, the workaround we came up with, with some help from Tahina Ramananandro from the F* team, makes use of the `peano_terminating` function introduced before.

Specifically, our workaround to prove this lemma is by explicitly establishing the logical connection between the divergent function `division` and the total function `division_terminating`. In the refinement predicate of `division`, we can intrinsically ensure that for any precondition, as long as the divisor for `division` is nonzero, its result is always equivalent to that of `division_terminating`. It follows that we can complete the proof by applying the total function `division_terminating` to the postcondition of the proof `peanoDivPf`.

Below we show the modified definition of the divergent `division` function as well as the type signature of the `peanoDivPf` lemma:

```
val division: a:peano -> b:peano ->
  Dv (c:peano{toNat b > 0 ==> c = div_terminating a b})

val peanoDivPf: m:peano -> n:peano{toNat n > 0} ->
  Lemma (ensures (division_terminating (mult m n) n == m))
```

Notice that this `peanoDivPf` above has exactly the same signature as that of the valid Peano division proof introduced at the beginning. Readers might wonder that the proof is verifying the terminating function `division_terminating`, not the `division` function we actually want to verify. In fact, the key point of the workaround is to prove intrinsically the equivalence of `division_terminating` and `division` under some logical conditions. Consider the modified divergent `division` function and notice the change we make on the function return type. Recall that the symbol `==>` stands for logical implies. We want to prove, intrinsically through the refinement predicate, that if the divisor `b` is greater than zero, then the result of the divergent `division` function is the same as that of the terminating `division_terminating` function. Therefore, given that we have a proof on the terminating `division_terminating` function, we indirectly have a proof for the `division` function that we want to verify.

4.2.2.3 Discussions

It is important to realize that the convoluted workaround presented in section 4.2.2.1 is a specific edge case that is not generally applicable. Even though the introduction of the intermediary total function `division_terminating` completes the proof, this workaround inevitably requires programmers to know the exact reason why the function they want to prove diverges. With that reason in mind, programmers first need to provide a counterpart of that divergent function and to find a decreasing metric to prove that it terminates. Then they have to come up with a logical connection between the divergent function and its terminating counterpart in the refinement predicate. Only after all these steps can they prove the desired theorem in the postcondition of a lemma.

Recall that termination analysis on a function is not accurate. The inaccuracy introduced generally gives rise to many functions whose termination is hard to prove. As a result, programmers generally can not know the reason why a function diverges, and hence cannot verify valuable properties on it.

In fact, many functions fall into the false negatives during termination analysis. Regardless of being total, they have to be marked as `Dv` in F*. Yet there are many important properties of these functions that we want to be able to verify. For example, both the big-step and multi-step evaluators on well-typed simply-typed lambda calculus terminate in theory and we want to verify the equivalence of these two evaluations (Consider the case when programmers optimize a compiler. It is crucial to make sure that the optimized

program evaluates to the exactly same result as what the original program evaluates to). However, it is really hard in practice, if not impossible, to find a termination metric to prove each of their termination. For this reason, both the big-step and multistep functions must be labeled with effect `Dv` and noun of them has a terminating counterpart. Therefore, we could not prove the equivalence of these two ways of evaluations in F*.

From our discussion with the F* developers, there might be some other intrinsic approaches to carry out proofs on functions of the `Dv` effect. One such approach is to find program invariants that generally hold for recursive functions. If we can decide the validity of these invariants, then by the relative completeness of Hoare Logic, there exist partially correct specifications for these divergent functions. Hoare Logic consists of Hoare triples, each of the form `P C Q`, where `P` and `Q` denote the precondition and postcondition correspondingly and `C` is the command being executed. Hoare Logic is relative complete since it can only derive the logical validity of specifications in the postcondition if the execution `C` is proven to terminate. Though this theory seems promising, to our knowledge, there is not yet any example proof of this kind in F*. After some careful studies of many related literatures, yet with our limited expertise in F* and the constrained time schedule, we did not come up with a working proof through program invariants. we believe that finding these invariants might still be practically restricted.

Another approach, based on its initial formalization in Adga, is called hereditary substitutions [10, 15]. This approach is brought up in the original F* paper as a potential workaround to prove that the big-step evaluator in Simply Typed Lambda Calculus terminates. However, the example provided on Github is unmaintained and to our understanding and in consultation with Eisenberg, is not generally applicable to any of our use cases.

In summary, we conclude that F* is in lack of a satisfiable approach in theorem proving on functions with effect `Dv` that may not terminate. Specifically, in divergent settings, extrinsic proofs are not yet supported whereas intrinsic proofs are practically limited.

5 Conclusion

This thesis has presented a thorough comparison between the two type systems in Dependent Haskell and F*. The comparison is focus on user interface and approaches towards potentially non-terminating functions. Our results on user interface suggest that as a newly-designed language, F* provides an elegant syntax with both dependent and refinement types, yet Dependent Haskell's syntax is intricate with its introduction of singletons for historical reasons. On the other hand, Dependent Haskell has many well-maintained libraries and well-supported documentations, but F* is still in lack of many resources. In terms of theorem proving, proofs in Dependent Haskell can be overwhelming especially for people new to program verification. Programmers in Dependent Haskell are expected to provide every single step of a proof. They generally encode a theorem as a GADT and specify it in a separate lemma. Yet, F* provides an intuitive and friendly user

interface for theorem proving, and generally requires less annotations than Dependent Haskell. Although, when verifications get larger and more complicated, F*'s interface gets harder to use quickly, gradually approaching the complexity of Dependent Haskell.

As for approaches towards potentially divergent functions, Dependent Haskell supports theorem proving on functions that may not terminate in a way that is similar to all other proofs. Yet, Dependent Haskell is weakly reliable, as the proof on type equality requires an additional verification at run time to determine its correctness. F*, on the other hand, strictly enforces its type system to be strongly reliable. However, at this stage, it does not yet provide an applicable solution to prove functions with effect `Dv`, that may not terminate.

From our analysis, we recommend using F* for small to medium sized verifications for its elegant syntax and user-friendly interface, but we highly suggest using Dependent Haskell for properties associated to functions that may not terminate. In terms of relatively larger verification requirements, both languages provide supports that are about equal, though neither of them is a perfect match at this stage. Specifically, Dependent Haskell, without any proof automation, might results in an enormous program and would be comparably more complicated to program with. On the other hand, F*, which heavily depends on the Z3 SMT solver for verifications, can get very time consuming as the contexts presented to Z3 get larger. More involved F* developments also introduce some intricacies such as indeterminate behaviors when programmers provide some bad SMT patterns.

We generally recommend F* to people new to program verification but recommend Dependent Haskell to professionals. Regarding to their programming interface, Dependent Haskell has a much better support. It bundles all its resources, including the compiler, Cabal library installation tool and etc, and allows programmers to access them through a single download. One big highlight worth mentioning is that all programs written in Dependent Haskell can be directly executed as a normal Haskell program. F*, on the other hand, requires installation of Z3 and the F* compiler separately. It is programmer's responsibility to set up all the path variables. Finally, after verification, programs in F* cannot be executed directly. It requires an additional step to extract F* programs into OCaml or F# for execution and also supports bootstrapping in C for performance concerns.

To conclude, we propose the following suggestions for each language as a potential future research direction:

Firstly, we advise Dependent Haskell to redesign its syntax, especially focusing on finding an alternative to singletons. We believe that Eisenberg's complete work on the full-fledged dependent types might be a good start, and we hope to see its implementation in GHC soon. Secondly, we suggest Dependent Haskell to consider refinement types, even though refinement types can be defined using dependent types. As through our experiences, refinement types are often more intuitive to work with. Finally, we recommend the F* development team to continue working on improving F*'s support for verifications on `Dv` functions that may not terminate.

Appendices

.1 Insersion Sort

.1.1 Dependent Haskell

```
{-# LANGUAGE IncoherentInstances, ConstraintKinds, TypeFamilies,
             TemplateHaskell, RankNTypes, ScopedTypeVariables, GADTs,
             TypeOperators, DataKinds, PolyKinds, MultiParamTypeClasses,
             FlexibleContexts, FlexibleInstances, UndecidableInstances #-}

module MyInsertionSort where

import Data.Kind (Type)
-- Mimics the Haskell Prelude, but with singleton types.
import Data.Singletons.Prelude
import Data.Singletons.SuppressUnusedWarnings
-- This module contains everything you need to derive your own singletons via
    Template Haskell.
import Data.Singletons.TH

-- Natural numbers, defined with singleton counterparts
$(singletons [d|
  data Nat = Zero | Succ Nat
  |])

-- A proof of the less-than-or-equal relation/constraint among naturals
data LeqProof :: Nat -> Nat -> Type where
  LeqZero :: LeqProof Zero a
  LeqSucc :: LeqProof a b -> LeqProof (Succ a) (Succ b)

-- A proof term asserting that a list of naturals is in ascending order
data NonDecProof :: [Nat] -> Type where
  AscEmpty :: NonDecProof '[]
  AscOne  :: NonDecProof '[n]
  AscCons :: LeqProof a b -> NonDecProof (b ': rest) -> NonDecProof (a ': b ':
      rest)

-- A proof term asserting that lst2 is the list produced when x is inserted
    (anywhere) into list lst1
data InsertionProof (x :: k) (lst1 :: [k]) (lst2 :: [k]) where
  -- Inserting x to the front of list l produces the list (x ': l)
  InsHere  :: InsertionProof x l (x ': l)
  -- If we get lst2 after inserting x into lst1, then we would get (y ': lst2)
      when we insert x into (y ': lst1)
  InsLater :: InsertionProof x lst1 lst2 -> InsertionProof x (y ': lst1) (y ':
      lst2)

-- A proof term asserting that lst1 and lst2 are permutations of each other
data PermutationProof (lst1 :: [k]) (lst2 :: [k]) where
```

```
  PermId  :: PermutationProof l l
  PermIns :: InsertionProof x lst2 lst2' -> PermutationProof lst1 lst2 ->
             PermutationProof (x ': lst1) lst2'

-- Here is the definition of functions about which we will be reasoning:
$(singletons [d|
  leq :: Nat -> Nat -> Bool
  leq Zero _         = True
  leq (Succ _) Zero = False
  leq (Succ a) (Succ b) = leq a b
  insert :: Nat -> [Nat] -> [Nat]
  insert n []    = [n]
  insert n (h:t) = if leq n h then (n:h:t) else h:(insert n t)
  insertionSort :: [Nat] -> [Nat]
  insertionSort []    = []
  insertionSort (h:t) = insert h (insertionSort t)
  |])

-- A lemma that states if sLeq a b is STrue, then we have a proof for (a :<=: b)
-- This is necessary to convert from the boolean definition of <= to the
    corresponding proof term to use in the type level
sLeq_true__le :: (Leq a b ~True) => SNat a -> SNat b -> LeqProof a b
sLeq_true__le a b = case (a, b) of
  (SZero, _)           -> LeqZero
  (SSucc a', SSucc b') -> LeqSucc (sLeq_true__le a' b')

-- A lemma that states if sLeq a b is SFalse, then we have a proof for (b :<=:
    a)
sLeq_false__nle :: (Leq a b ~False) => SNat a -> SNat b -> LeqProof b a
sLeq_false__nle a b = case (a, b) of
  (SSucc _, SZero)     -> LeqZero
  (SSucc a', SSucc b') -> LeqSucc (sLeq_false__nle a' b')

-- A lemma that states that inserting into an ascending list produces an
    ascending list
insert_ascending :: forall n lst. SNat n -> SList lst -> NonDecProof lst ->
    NonDecProof (Insert n lst)
insert_ascending n lst ascPf_lst = case ascPf_lst of
  AscEmpty              -> AscOne -- If lst is empty, then we're done
  AscOne                -> case lst of -- If lst has one element...
    SCons h SNil -> case sLeq n h of -- then check if n is <= h
      STrue  -> AscCons (sLeq_true__le n h) AscOne -- if so, we're done
      SFalse -> AscCons (sLeq_false__nle n h) AscOne-- if not, we're done
  AscCons hLeqH2 ascPf_t -> case lst of -- Otherwise, if lst is more than one
      element...
    SCons h t -> case sLeq n h of -- then check if n is <= h
      STrue -> AscCons (sLeq_true__le n h) ascPf_lst -- if so, we're done
      SFalse -> case sLeq_false__nle n h of -- if not, things are harder...
        hLeqN -> case insert_ascending n t ascPf_t of
          ascPf_nt -> case t of
            SCons h2 _ -> case sLeq n h2 of
```

```
            STrue -> AscCons hLeqN ascPf_nt -- since we have h <= n and
                t(h2:_) is ascending
            SFalse -> AscCons hLeqH2 ascPf_nt -- since we have h <= h2 and
                t(h2:_) with n inserted is ascending

-- A lemma that states that the result of an insertion sort is in ascending
    order
nondec_pf :: SList lst -> NonDecProof (InsertionSort lst)
nondec_pf lst = case lst of
  SNil     -> AscEmpty -- if the list is empty, we're done
  -- otherwise, we recur to find that insertionSort on t produces an ascending
      list, and then we use the fact that inserting into an ascending list
      produces an ascending list
  SCons h t -> case nondec_pf t of
    ascPf_t -> insert_ascending h (sInsertionSort t) ascPf_t

-- A lemma that states that inserting n into lst produces a new list with n
    inserted into lst.
insert_insertion :: SNat n -> SList lst -> InsertionProof n lst (Insert n lst)
insert_insertion n lst =
  case lst of
    SNil    -> InsHere -- if lst is empty, we're done (by def of insert)
    SCons h t -> case sLeq n h of -- otherwise, is n <= h? (work with
        Singletons)
      STrue -> InsHere -- if so, we're done (by def of insert)
      SFalse -> InsLater (insert_insertion n t) -- otherwise, recur

-- A lemma that states that the result of an insertion sort is a permutation of
    its input
permutation_pf :: SList lst -> PermutationProof lst (InsertionSort lst)
permutation_pf lst = case lst of
  SNil     -> PermId -- if the list is empty, we're done
  -- otherwise, we wish to use PermIns. We must know that t is a permutation of
      the insertion sort of t and that inserting h into the insertion sort of t
      works correctly:
  SCons h t ->
    case insert_insertion h (sInsertionSort t) of
      insertionPf -> PermIns insertionPf (permutation_pf t)

-- A theorem that states that the insertion sort of a list is both ascending
    and a permutation of the original
insertionSort_pf :: SList lst -> (NonDecProof (InsertionSort lst),
                                  PermutationProof lst (InsertionSort lst))
insertionSort_pf slst = (nondec_pf slst,
                         permutation_pf slst)
```

.2 Peano Division

.2.1 Dependent Haskell

```
{-# LANGUAGE GADTs, TypeInType, ScopedTypeVariables, StandaloneDeriving,
             TypeFamilies, TypeOperators, AllowAmbiguousTypes, TypeApplications,
             UndecidableInstances, DataKinds #-}

module PeanoDivisionDH where

import Data.Kind ( Type )
import Data.Type.Equality ( (:~:)(..) )
import Prelude hiding ( div )

data Peano where
  Zero :: Peano
  Succ :: Peano -> Peano
deriving (Eq, Ord, Show)

type family (a :: Peano) + (b :: Peano) :: Peano where
  Zero + b     = b
  (Succ a) + b = Succ (a + b)
infix 6 +

type family (m :: Peano) - (n :: Peano) :: Peano where
  Zero - _            = Zero
  m - Zero            = m
  (Succ m) - (Succ n) = m - n
infix 6 -

type family (a :: Peano) * (b :: Peano) :: Peano where
  Zero * _     = Zero
  (Succ a) * b = a * b + b
infix 7 *

type family (a :: Peano) / (b :: Peano) :: Peano where
  Zero / _ = Zero
  m / n    = (Succ Zero) + (m - n) / n
infix 7 /

data SNat n where
  SZero :: SNat Zero
  SSucc :: SNat n -> SNat (Succ n)
deriving instance Show (SNat n)

plus :: SNat m -> SNat n -> SNat (m + n)
plus SZero n     = n
plus (SSucc m) n = SSucc (plus m n)

minus :: SNat m -> SNat n -> SNat (m - n)
minus SZero _              = SZero
minus m SZero              = m
minus (SSucc m) (SSucc n) = minus m n

mult :: SNat m -> SNat n -> SNat (m * n)
```

```
mult SZero _     = SZero
mult (SSucc m) n = plus (mult m n) n

plusZero :: SNat m -> m + Zero :~: m
plusZero SZero     = Refl
plusZero (SSucc m) = case (plusZero m) of Refl -> Refl

plusSucc :: SNat m -> SNat n -> m + (Succ n) :~: Succ (m + n)
plusSucc SZero _     = Refl
plusSucc (SSucc m) n = case (plusSucc m n) of Refl -> Refl

--   (m + (Succ n)) - (Succ n)
-- = Succ (m + n) - (Succ n), by plusSucc
-- = (m + n) - n, by definition of (-)
-- = m, by inductive hypothesis
plusMinus :: SNat m -> SNat n -> (m + n) - n :~: m
plusMinus m SZero     = case (plusZero m) of Refl -> Refl
plusMinus m (SSucc n) =
   case (plusSucc m n) of Refl
-> case (plusMinus m n) of Refl
-> Refl

--   (Succ m * Succ n) / Succ n
-- = (m * Succ n + Succ n) / Succ n, by definition of (*)
-- = Succ (m * Succ n + n) / Succ n, by plusSucc
-- = Succ Zero + (Succ (m * Succ n + n) - Succ n) / Succ n, by definition of (/)
-- = Succ Zero + ((Succ (m * Succ n) + n) - Succ n) / Succ n, by definition of
    (+)
-- = Succ (((Succ (m * Succ n) + n) - Succ n) / Succ n), by definition of (+)
-- = Succ (((m * Succ n) + Succ n - Succ n) / Succ n), by plusSucc (same as
    above, opposite direction)
-- = Succ ((m * Succ n) / Succ n), by plusMinus
-- = Succ m, by inductive hypothesis
peanoDivisionPf :: SNat m -> SNat (Succ n) -> (m * (Succ n)) / (Succ n) :~: m
peanoDivisionPf SZero _                    = Refl
peanoDivisionPf (SSucc m) succ_n@(SSucc n) =
   case plusMinus (mult m succ_n) succ_n of Refl
-> case plusSucc (mult m succ_n) n of Refl
-> case peanoDivisionPf m succ_n of Refl
-> Refl
```

.2.2 F*

```
module PeanoDivision

type peano: Type =
  | Zero : peano
  | Succ : peano -> peano

val toNat: peano -> Tot nat
let rec toNat a = match a with
```

```
  | Zero    -> 0
  | Succ a' -> 1 + toNat a'

val max: int -> int -> Tot int
let max a b = if a > b then a else b

val plus: a:peano -> b:peano -> Tot (c:peano)
let rec plus a b = match a with
  | Zero    -> b
  | Succ a' -> Succ (plus a' b)

val minus: a:peano -> b:peano ->
  Tot (c:peano{toNat c = max (toNat a - toNat b) 0})
let rec minus a b = match a with
  | Zero    -> Zero
  | Succ a' -> match b with
    | Zero    -> Succ a'
    | Succ b' -> minus a' b'

val mult: a:peano -> b:peano -> Tot (c:peano)
let rec mult a b = match a with
  | Zero    -> Zero
  | Succ a' -> plus (mult a' b) b

val division_terminating: a:peano -> b:peano{toNat b > 0} -> Tot (c:peano)
  (decreases (toNat a))
let rec division_terminating a b = match a with
  | Zero   -> Zero
  | Succ _ -> if (toNat a < toNat b) then Zero
              else plus (division_terminating (minus a b) b) (Succ Zero)

val division: a:peano -> b:peano -> Div (c:peano)
  (requires True)
  (ensures (fun y -> toNat b > 0 ==> y == div_terminating a b))
let rec division a b = match a with
 | Zero   -> Zero
 | Succ _ -> if (toNat a < toNat b) then Zero
             else let denom = (division (minus a b) b)
                  in plus denom (Succ Zero)

val plusZero: m:peano -> Lemma (ensures plus m Zero = m)
let rec plusZero m = match m with
  | Zero   -> ()
  | Succ m -> plusZero m

val plusSucc: m:peano -> n:peano -> Lemma (ensures plus m (Succ n) = Succ (plus
    m n))
let rec plusSucc m n = match m with
  | Zero   -> ()
  | Succ m -> plusSucc m n
```

```
val plusMinus: m:peano -> n:peano -> Lemma
  (requires True)
  (ensures minus (plus m n) n = m)
let rec plusMinus m n = match n with
  | Zero   -> plusZero m
  (*
    minus (plus a (Succ b')) (Succ b')
  = minus (Succ (plus a b')) (Succ b') by plusSucc a b'
  = minus (plus a b') b'
  = a by minusPlus a b'
  *)
  | Succ n -> match plusSucc m n with () -> plusMinus m n

val peanoDivPf: m:peano -> n:peano{toNat n > 0} -> Lemma
  (requires True)
  (ensures (div (mult m n) n == m))
let rec peanoDivPf m n = match m with
  (*
    div (mult Zero b) b = div Zero b = Zero
  *)
  | Zero   -> ()
  (*
    div (mult (Succ a') b) b
  = div (plus (mult a' b) b) b
  = plus (div (minus (plus (mult a' b) b) b) b) (Succ Zero)
  = plus (div (mult a' b) b) (Succ Zero) by plusMinus (mult a' b) b
  = plus a' (Succ Zero) by peanoDivPf a' b
  = Succ (plus a' Zero) by plusSucc a' Zero
  = Succ a' by plusZero a'
  *)
  | Succ a' -> plusMinus (mult a' b) b;
               peanoDivPf a' b;
               plusSucc a' Zero;
               plusZero a'
```

References

[1] The 2017 Top Programming Languages. https://spectrum.ieee.org/computing/software/the-2017-top-programming-languages. Accessed: 2018-05-15.

[2] Verified programming in F* - A Tutorial. https://www.fstar-lang.org/tutorial/tutorial.html#sec-the-st-effect. Accessed: 2018-05-15.

[3] John N Buxton and Brian Randell. *Software Engineering Techniques: Report on a Conference Sponsored by the NATO Science Committee*. NATO Science Committee; available from Scientific Affairs Division, NATO, 1970.

[4] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[5] Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[6] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[7] Richard A. Eisenberg. *Dependent Types in Haskell: Theory and Practice*. PhD thesis, University of Pennsylvania, 2016.

[8] Richard A. Eisenberg and Stephanie Weirich. Dependently Typed Programming with Singletons. In *ACM SIGPLAN Haskell Symposium*, 2012.

[9] Adam Gundry. *Type Inference, Haskell and Dependent Types*. PhD thesis, University of Strathclyde, 2013.

[10] Chantal Keller and Thorsten Altenkirch. Normalization by hereditary substitutions. *proceedings of Mathematical Structured Functional Programming*, 2010.

[11] K. Rustan M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'16, pages 348–370. Springer Berlin Heidelberg, 2010.

[12] K Rustan M Leino. Automating Induction with an SMT solver. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 315–331. Springer, 2012.

[13] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[14] Benjamin C Pierce. *Types and Programming Languages*. MIT press, 2002.

[15] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. De-

pendent Types and Multi-monadic Effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16. ACM, 2016.

[16] Brent A Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving Haskell a Promotion. In *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation*, pages 53–66. ACM, 2012.